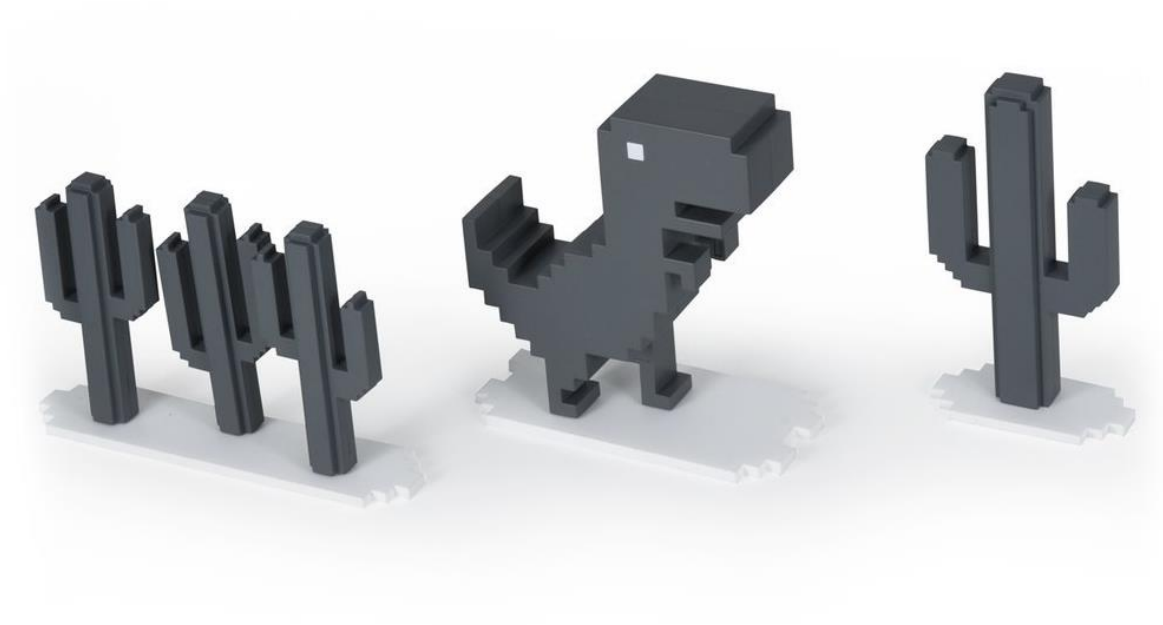
РУКОВОДСТВО ПО НАПИСАНИЮ ИГРЫ НА C++



Новосибирск
2019

ОГЛАВЛЕНИЕ.

ОСНОВНАЯ ИНФОРМАЦИЯ.

1. Используемые библиотеки.

Для разработки игры вы будете использовать следующие библиотеки:

- ✓ **SDL v2.0.9** – базовая библиотека для графики.
- ✓ **SDL2_image v2.0.5** – расширение, позволяющее работать с изображениями различных форматов.
- ✓ **SDL2_mixer v2.0.4** – расширение для работы со звуком.
- ✓ **SDL2_ttf v2.0.15** – расширение для работы с текстом.

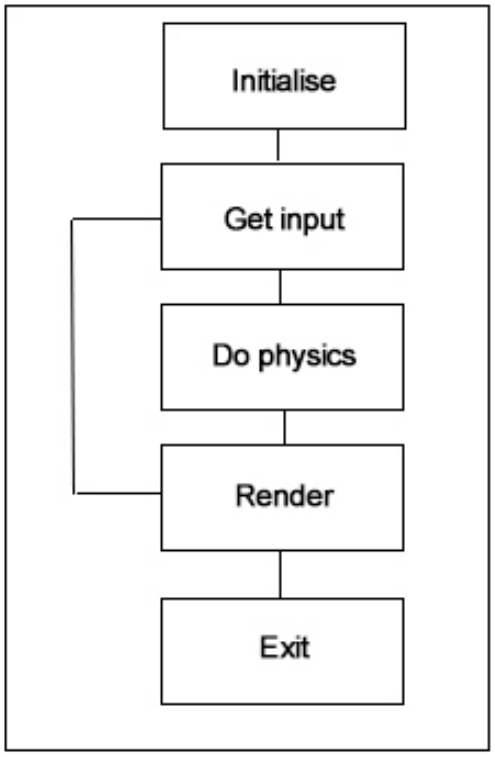
- ✓ **linalg** – only-header библиотека с линейной алгеброй**.** Подробнее о библиотеке написано здесь.

- ✓ **cpp-httplib** – only-header библиотека для работы с сетью по протоколу HTTP/HTTPS. Подробнее о библиотеке написано здесь.

- ✓ **Box2D** – физический движок. Документацию можно найти здесь.

2. Основные понятия.

Подавляющее большинство игр состоит, в основном из 5 функций, вызываемых в главном цикле работы программы, которые и обрабатывают весь игровой процесс. Краткое описание каждой из них:



Функция инициализации
Обрабатывает все загрузки данных, будь то текстуры, карты, персонажи, или любые другие (аудио\видео к примеру).

Обработчик событий
Обрабатывает все входящие сообщения от мыши, клавиатуры, джойстиков, или других устройств, либо программных таймеров.

Игровой цикл
Обрабатывает все обновления процесса, такие как смена координат толпы противников, движущихся по экрану и стремящихся уменьшить ваше здоровье, или что-то иное (расчет появления бонусов, обнаружение столкновений).

Отрисовка сцен
Занимается отображением рассчитанных и подготовленных в предыдущей функции сцен на экран, и никаких манипуляций с данными производить не обязана.

Очистка памяти
Просто удаляет все загруженные ресурсы (карты, изображения, модели) из ОЗУ, и обеспечивает корректное завершение игры.

Как видно по рисунку, сначала происходит инициализация, затем в каждой итерации цикла происходит обработка событий, манипуляция с данными и, соответственно, отрисовка, после чего программа корректно завершается, выгружая данные. Иногда для создания игры обработка событий не требуется, но необходима, когда вы хотите, чтобы пользователь мог манипулировать данными (например, перемещением персонажа).

ИНСТРУКЦИЯ ПО ИСПОЛЬЗОВАНИЮ ИГРОВОГО ФРЕЙМВОРКА.

1. Класс BaseGame и класс Game.

Класс `BaseGame` *(находится: /framework/base_game/)* – является основой нашей игры. В его контракте определены следующие поля и методы:

```
class BaseGame : public InputHandler {
public:
    BaseGame();
    float getWidth() const;                 //Ширина игрового окна в пикселях
    float getHeight() const;                //Высота игрового окна в пикселях
    SDL_Renderer* getRenderer() const;      //Получить рендерер нашей игры
    bool isRunning() const;                 //Проверяем, работает ли игра (в начале true)
    void quit();                            //Выход из игры – running = false;
    virtual void preload() = 0;             //Метод для инициализации
    virtual void render() = 0;              //Метод для отрисовки игрового мира
    virtual void update(float dt) = 0;      //Метод для обновления игрового мира
    virtual void handleEvents() = 0;        //Метод для обработки событий
    virtual ~BaseGame();
private:
    bool running = true;
    std::unique_ptr<Window> window;         //Указатель на окно
};
```

А теперь давайте посмотрим на использование класса `Game`, наследника `BaseGame`, в *main.cpp*:

```
int SDL_main(int argc, char ** argv) {
    if (initialize() != EXIT_SUCCESS)
        return EXIT_FAILURE;
    {
        Game game;
        game.preload();
        Uint32 ticks = 0;
        while (game.isRunning()) {
            float dt = (SDL_GetTicks() - ticks) / 1000.0f;    //Время в секундах
            ticks = SDL_GetTicks();
            game.render();
            game.update(dt);
            game.handleEvents();
        }
    }
    finalize();
    return 0;
}
```

Итак, что здесь происходит?

Мы создаем объект класса `Game`, выполняем инициализацию нашей игры (метод `preload`). Далее мы входим в цикл:

- Считаем время, прошедшее с последней итерации, оно нам пригодится.
- Выполняем отрисовку игрового мира
- Обновляем игровой мир *(Передаем время в качестве параметра, чтобы выполнять изменения, зависящие от времени. Например, необходимо посчитать координату тела, движущегося со скоростью: x += Vx * dt;)*
- Обрабатываем события.

Так продолжается до тех пор, пока метод `isRunning()` не вернет `false`. А это произойдет, если вызвать метод `quit()`.

Функции `initialize()` и `finalize()` нас не интересуют. Пока что достаточно знать, что первая инициализирует работу библиотеки **SDL**, а вторая, наоборот, завершает работу.

2. События и их обработка. Класс InputHandler.

Для обработки событий используется класс *InputHandler*. Основной метод:
`void InputHandler::onEvents(const SDL_Event &event);`
Он анализирует переданное ему событие, и вызывает один из определенных в классе обработчиков. Класс `BaseGame` наследуется от `InputHandler`.

Итак, как же обрабатывать события?

```
//В этом методе просматриваем все события и вызываем обработчик для события.
void Game::handleEvents() {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        onEvents(event);
    }
}
//Обработчик события нажатия клавиши
void Game::onKeyDown(const SDL_Keysym &sym) {
    if (sym.sym == SDLK_ESCAPE)
       //.........
}
//Обработчик события закрытия окна
void Game::onQuit() {
    quit();
}
```

Метод `onEvents()` уже определен в классе `InputHandler`, методы `onKeyDown()` и `onQuit()` из него же мы переопределили в классе `Game`.
Так как все методы в `InputHandler` помечены как виртуальные, метод `onEvents()` вызывает методы-обработчики именно класса `Game`.

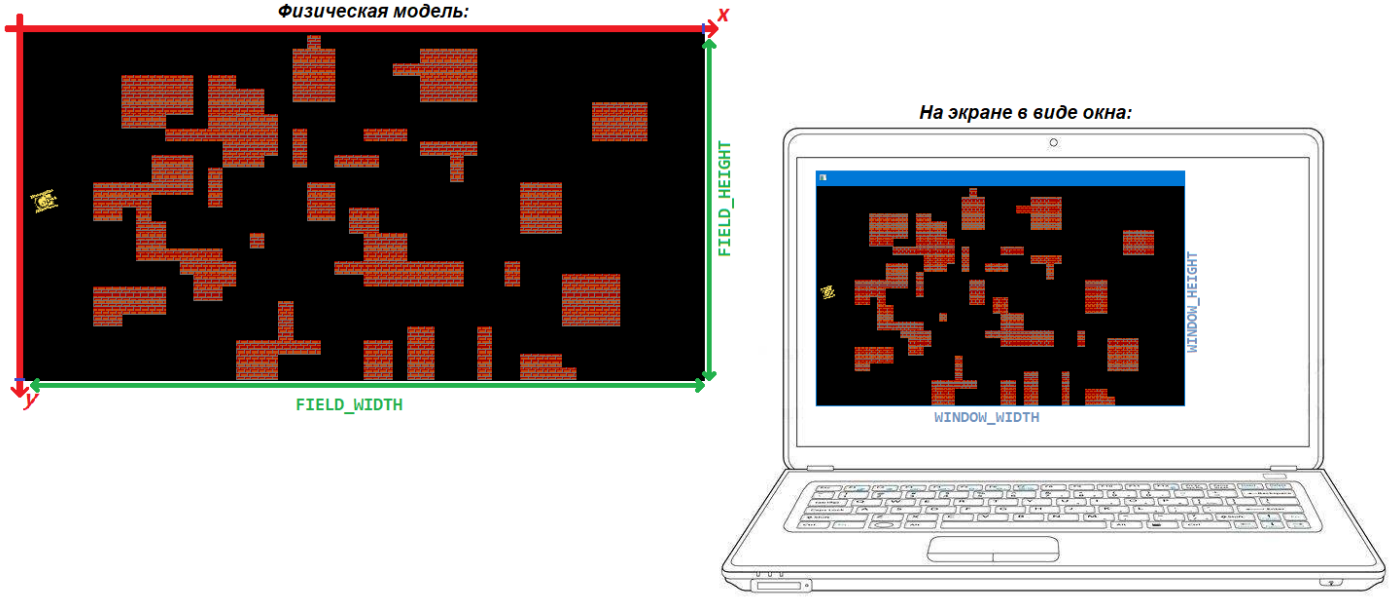
```
class Game : public BaseGame {
public:
    //Как-то так:
    void onKeyDown(const SDL_Keysym &sym);
    void onQuit();
};
```

3. Конфигурация игры. Физическая и графическая система отсчета.

В файле *config/configuration.cpp* вы можете задать `WINDOW_FLAGS` – параметры игрового окна. Список всех флагов здесь.

Чтобы задать размер окна вашей игры, откройте файл *base_game/base_game.cpp* и измените параметры, передаваемые в конструктор `Window`. Можно получить разрешение экрана при помощи функции `getDesktopMetrics`.

Теперь самое время разобраться связи графических и физических координат. Посмотрите на картинку ниже:



Как можно догадаться, физические величины вашей системы отсчета не всегда совпадут с графическими. Поэтому необходимо их преобразовать. Для этого в *config* присутствуют две функции конвертации. Первая преобразует такие величины как длина, ширина и тд, а вторая – координаты (задаются в структуре `SDL_Point` полями x и y):

```
//field_width и field_height - физические размеры вашего поля, для которого
//нужно выполнить пересчет координат:

int convert(SDL_Renderer *renderer,
    int field_width,
    int field_height,
    double value);

SDL_Point convert(SDL_Renderer *renderer,
    int field_width,
    int field_height,
    const SDL_Point &point);
```

Почему две функции? Так как соотношение сторон физической модели поля может не совпадать с соотношением сторон окна, в котором оно рисуется, то в окне помимо изображения поля будут черные полосы – места, где ничего не отрисовывается. Соответственно начало системы координат физической модели будет находиться не там же, где начало системы координат окна **SDL** – будет сдвиг по горизонтали или вертикали. Второй конвертер это учитывает, а первый – просто умножает скалярную величину на коэффициент.

Совет. Так как система координат для окна **SDL** считается от верхнего левого угла, ось OX направлена вправо, ось OY направлена вниз, а углы считаются по часовой стрелки относительно OX, то советуем вам строить физическую модель так же.

В функции `initialize()`, как говорилось ранее, инициализируется работа **SDL**. При желании, мы можете поменять макросы и флаги, используемые в init-функциях, чтобы оставить только необходимое вам.

4. Игровые сущности. Класс Entity. Класс World.

В любой игре должны быть игровые сущности. В *entity/entity.h* определена структура вашей сущности. Начнем с основ:

Класс *GraphicObject.*

- `void update(float dt)` – для обновления подсистемы во времени. Например, у сущности анимированная текстура из нескольких кадров, которые меняются через какое-то время.
- `void draw(SDL_Renderer *renderer)` – для отрисовки сущности.

Класс *PhysicalObject* – в этом классе будет определена вся физика вашей сущности.

- Виртуальный метод `checkCollid(PhysicalObject *other)` необходим для проверки и обработки столкновений между нашим объектом и каким-то другим, переданным в метод.
- Еще один важный метод, определенный в физике – `TYPE getType() const`.
  Он возвращает тип вашего объекта. Объект может быть двух типов – статический и динамический. В играх некоторые сущности обычно перемещаются, и могут сталкиваться с другими. Так вот, сущности, которые могут перемещаться в пространстве являются DYNAMIC, а которые нет – STATIC.
- `void update(float dt)` – обновляем физику сущности. Например, перемещаем.

Класс *SoundObject* – в нем не определено ничего, кроме *update(dt).* Как его реализовывать – тут полная творческая свобода.

Как можете видеть класс *Entity* содержит в себе указатели на физическую, графическую и звуковую подсистемы. Это сделано для удобства, чтобы не городить один огромный класс, который отвечает и за физику, и за графику и тд.
Обратите внимание на метод `bool isAlive()`. В процессе игры может выясниться, что сущность умирает, и больше ее не нужно обрабатывать. Так вот, сущность будет

обрабатываться до тех пор, пока этот метод возвращает `true`.

Один из способов реализации этого метода:

```
bool Tank::isAlive() const {
   return hp > 0;  //в процессе игры hp может уменьшатся, что и приводит к гибели сущности
}
```

Для создания сущностей используйте функцию createEntity:

```
//Функция возвращает std::unique_ptr с объектом:
auto tank = createEntity<Tank>(/* args */);

//Можете использовать как указатель:
tank->initGraphics();
tank->initSounds();
```

Теперь перейдем к классу `World`.
World – по сути, хранилище всех ваших игровых сущностей. В нем определены методы:

- `void update(float dt)` – делает обновление сущностей, обрабатывает коллизии.
- `void draw(SDL_Renderer *renderer)` – отрисовка всех сущностей
- `void addEntity(Entity&& entity)` – добавление сущности в *World*

Entity содержит указатель на объект класса `World` – т.к. может понадобиться из метода сущности обратиться к world, который ее содержит. Например:

```
//пуля порождается внутри метода shoot, но ее
//необходимо добавить в мир
void Tank::shoot() {
    //.................
    auto bullet = createEntity<Bullet>(>(/* args */));
    bullet->initGraphics();


    //std::unique_ptr можно только перемещать, после добавления в World
    //вы больше не сможете использовать его:
    getWorld()->addEntity(std::move(bullet));

}
```

И да, не нужно беспокоиться об очистке памяти. Память, будет очищена автоматически.

5. Загрузка звуков и картинок.

5.1. Картинки

Для загрузки картинок можно воспользоваться классом `TextureManager`. Этот класс является синглтоном. Поэтому для работы с ним нужно воспользоваться статическим

методом `TextureManager* getInstance()`. Для загрузки картинки воспользуйтесь следующим методом:

```
void load(SDL_Renderer *renderer, const std::string &id,
      const std::string &path, int frame_width, int frame_height)
```

`renderer` – ваш рендерер
`id` – идентификатор, с помощью которого вы потом будете получать текстуру
`path` – путь до файла (может быть как относительным, так и абсолютным)
`frame_width`, `frame_height` – ширина и высота одного кадра (что такое кадр – будет написано дальше)

Для того чтобы получить текстуру, воспользуйтесь методом:
`Texture get(const std::string &id) const`
который вернет объект класса *Texture*. Его можно использовать в качестве `SDL_Texture*` а так же получить размер кадра при помощи методов `getWidth()` и `getHeight()`.

Пример использования:

```
void preload(SDL_Renderer *renderer) override {
    TextureManager::getInstance()->load(renderer, "tank", "textures/tank.png", 175, 144);
}
//Поддерживаются любые форматы картинок

void draw(SDL_Renderer *renderer) {
    Texture texture = TextureManager::getInstance()->get("tank");
    int frame_width = texture.getWidth();
    int frame_height = texture.getHeight();
    //..............

    SDL_RenderCopyEx(renderer, texture, &src_rect, &dest_rect, 0, NULL, SDL_FLIP_NONE);
}
```

Так же можете указать директорию по умолчанию, из которой будут браться текстуры:

```
//Относительный путь:
TextureManager::getInstance()->setDefaultPath("../textures");

//Или абсолютный:
TextureManager::getInstance()->setDefaultPath("C:/Game/textures");

//И далее файлы будут искаться в указанной директории:
TextureManager::getInstance()->load(renderer, "tank", "tank.png", 175, 144);
```

Что считать кадром? Если у вас будут анимированные объекты в игре, то лучше всего хранить кадры в виде одной картинки, один за другим:

Так вот, для удобства при загрузке картинки лучше указывайте размер не всей картинки, а одного кадра, чтобы можно было сделать что-то такое:

```
SDL_Rect src_rect {
    getCurrentFrame() * frame_width,
    0, frame_width, frame_height
};
//В библиотеке SDL можно взять часть картинки, определив src_rect
//В данном случае при увеличении возвращаемого getCurrentFrame() значения мы берем новый
//кадр
```

5.2. Звук

Для загрузки звука используйте класс `SoundManager`. Этот класс является шаблонным, и определен для двух типов – `Mix_Chunk` и `Mix_Music`. С этим классом все гораздо проще, ниже пример его использования:

```
void loadMusic() {   //Можно загружать .flac .mp3 .ogg и др.
    SoundManager<Mix_Chunk>::getInstance()->setDefaultPath("../sounds");
    SoundManager<Mix_Chunk>::getInstance()->load("shoot", "shoot.ogg");
    SoundManager<Mix_Music>::getInstance()->load("music", "music.mp3");
}
//Получить:
void performEffect(const std::string& name, int channel, int loops) {
    Mix_Chunk* chunk = SoundManager<Mix_Chunk>::getInstance()->get(name);
    Mix_PlayChannel(channel, chunk, loops);
}
```

5.3. Очистка памяти.

Все загруженные файлы должны быть очищены, **причем это должно произойти до того, как в функции WinMain будет вызван** `finalize()`. Каждый из этих классов содержит статический метод `shutdown()`.

Настоятельно рекомендуем поместить его тут:

```
int SDL_main(int argc, char ** argv) {
    //...................
    //после основного цикла с игрой, НО перед finalize():
    AnimationManager::shutdown();  //про менеджер анимаций в следующем разделе
    TextureManager::shutdown();
    SoundManager<Mix_Music>::shutdown();
    SoundManager<Mix_Chunk>::shutdown();
    finalize();
    return EXIT_SUCCESS;
}
```

6. Анимация. Менеджер спрайтов.

6.1. Класс Animation.

```
class Animation : public GraphicObject {
public:
    Animation(float framerate_, int frames_count_, bool repeat_ = false);

    //для обновления текущего кадра:
      void update(float dt) override;
    //вернуть номер текущего кадра (нумерация начинается с 0)
    //если кадры закончились – вернет -1:
      int getCurrentFrame() const;
    //вернуть общее количество кадров:
      int getFramesCount() const;
    //Установить число кадров. Если номер текущего кадра больше либо равен count
    //то текущий кадр становится равным 0:
      void setFramesCount(int count);
    //Следующий кадр. Возвращает номер нового кадра. Если кадры закончились – вернет -1
      int nextFrame();
};
```

У анимации есть 3 параметра: число кадров в секунду, количество этих кадров, и флаг – повторять анимацию или нет.

Если у вас игровая сущность и вы хотите сделать ей анимированную текстуру с помощью этого класса, то `repeat = true` (последовательность кадров будет повторяться сначала). Если же вы хотите использовать этот класс как базовый для анимаций, которые после последнего кадра больше не отрисовываются, то воспользуйтесь значением по умолчанию.

Этот класс содержит методы для управления кадрами. Наиболее важный из них – `getCurrentFrame()` – он уже был использован ранее, в **разделе 5.1**. С помощью него вы можете узнавать номер кадра.

По сути, этот базовый класс – просто счетчик кадров, в его реализациях вам нужно, пользуясь номерами кадров, определить метод draw(). Метод update() тоже можно переопределить, если хотите изменять кадры как-нибудь по-другому, например через nextFrame() или setFramesCount(). Пример:

```
//Анимация танка состоит из 2 кадров, и если танк движется, то мы отрисовываем оба по
//очереди, если же нет, то только 1 кадр
void TankGraphics::update(float dt) {
    AnimatedBasicGraphics::update(dt);  //обновляем счетчик в базовом классе
    if (physics->getMove() == MOVE_NONE && physics->getRotation() == ROTATE_NONE)
        setFramesCount(static_frames);
    else
        setFramesCount(move_frames);
}
```

6.2. Класс SpriteManager.

Далее будем называть анимированные графические объекты **спрайтами**.

Спрайт – наследник класса `Animation`. Спрайт – не сущность, и никак с ними не взаимодействует. Поэтому для них создан отдельный класс `SpriteManager`, их обрабатывающий.

```
class SpriteManager : public GraphicObject {
public:
    static SpriteManager* getInstance();
    static void shutdown();


    template<class Animation_type, class... Args>
    void addSprite(Args&&... args);

    void draw(SDL_Renderer *renderer) override;  //нарисовать все спрайты
    void update(float dt) override;              //обновить все спрайты
};
```

Шаблонный метод `addSprite` – нужен для добавления спрайтов:

```
//Параметр шаблона – тип анимации, в качестве параметров – аргументы конструктора спрайта
SpriteManager::getInstance()->addSprite<Explosion>(/* args */);
```

Для очистки памяти не забудьте вызвать метод `void shutdown()`, обязательно перед `finalize()`.

7. Уровни. Менеджер уровней.

7.1. Класс Level.

Класс `Level` (*framework/level_manager/level.h*):

```
class Level : public InputHandler {
    bool running = true;
    std::unique_ptr<World> world;
public:
    Level() : world(new World) {}

    bool isRunning() const {
        return running;
    }

    virtual void preload(SDL_Renderer *renderer) {/* here you can load files */}

    virtual void clear() {/* here you can free unnecessary files */}

    virtual void update(float dt) {
        world->update(dt);
    }
```

```
    virtual void draw(SDL_Renderer *renderer) {
        world->draw(renderer);
    }

    virtual void quit() {
        running = false;
    }

    World* getWorld() const {
        return world.get();
    }

    virtual ~Level() {}
};
```

Первое, на что стоит обратить внимание – уровень порождает объект класса `World` при создании и для доступа к нему из наследников сделан метод `World* getWorld() const`.

Так же класс `Level` является наследником `InputHandler`, так что имеет смысл обрабатывать события внутри самого уровня. Так же имеются методы `preload()` и `clear()` для загрузки каких-либо файлов и очистки в конце.

Идея для игры: можете сделать `class Menu: public Level` и по нажатию кнопки “PLAY”, изнутри вызывается метод `quit()`, что приводит к запуску следующего уровня – уже самой игры. О запуске уровней написано ниже.

7.2. Класс LevelManager

Для управления уровнями был создан класс `LevelManager` *(framework/level_manager/level_manager.h):*

```
class LevelManager : public InputHandler {
public:
    static LevelManager& getInstance(); //   <-синглтон

    //Указать уровни в виде параметра шаблона, можно указать сразу все уровни
    template <class... level_types> void addLevel();

    //Обязательно вызовите эту функцию, или в конструкторе Game или в preload
    //и передайте ей в качестве параметра рендерер при помощи getRenderer()!
    void start(SDL_Renderer *renderer_);

    //is_running = false
    void quit();

    //return is_running – обязательно проверяйте в методах класса Game!
    bool isRunning();

    void render();
    void update(float dt);                  //Обрабатываем и отрисовываем текущий уровень
    void onEvents(const SDL_Event &event);
```

```
    //Установить правило смены уровней
    void setLevelChangeStrategy(LevelChangeStrategy *strategy);

    //Поставить загрузочный экран
    void setLoadingScreen(LoadingScreen *screen);
};
```

Итак, у нас есть менеджер – он конструирует следующий уровень при завершении предыдущего.

Перестает же работать менеджер после завершения последнего уровня, или после вызова метода `quit()`. Соответственно, метод `isRunning()` возвращает `true` до тех пор, пока менеджер не завершил свою работу.

Теперь давайте поговорим про `setLevelChangeStrategy()`. Представим, что наш текущий уровень уже завершился, но мы не хотим, чтобы следующий запускался сразу же (именно так ведет себя менеджер по дефолту). Например, мы хотим запустить следующий уровень через какое-то время или по нажатию клавиши. Для этого и создан этот метод. Он принимает параметром указатель на объект класса `LevelChangeStrategy`:

```
class LevelChangeStrategy {
public:
    bool isReadyToChange(float dt) const;
    bool isReadyToChange(const SDL_Event &event) const;
    void setStrategy(const std::function<bool(float dt)> &strategy);
    void setStrategy(const std::function<bool(const SDL_Event&)> &strategy);
};
```

Итак, следующий уровень запускается, если одна из функций `isReadyToChange()` вернет `true`. Соответственно, вы можете определить любую из них, или обе (ниже будут примеры).

Если вы установили правило на смену уровня, значит, скорее всего, он не сменяется мгновенно. А значит, имеет смысл воспользоваться еще одним методом:
`void setLoadingScreen(LoadingScreen *screen)`

А вот и сам класс `LoadingScreen`:

```
class LoadingScreen {
public:
    virtual void update(float dt) = 0;
    virtual void draw(SDL_Renderer *renderer) = 0;

    virtual ~LoadingScreen() {}
};
```

Получается, в наследнике вы можете определить метод `draw()` – пусть рисует красивую заставку, например. Можете и определить метод `update()`, если хотите, чтобы ваша изображение было динамичным, например, полоска загрузки и тд.

Теперь рассмотрим применение всех этих классов на примерах:

```
Game::Game() : BaseGame(), level_manager(LevelManager::getInstance()) {
    //можете добавить референс на LevelManager в качесте поля Game, чтобы каждый раз
    //не вызывать getInstance()
    LevelChangeStrategy *strategy = new LevelChangeStrategy{};
    strategy->setStrategy(     //Запуск уровня по нажатии “Z”
        [](const SDL_Event &event) {
            if (event.type == SDL_KEYDOWN)
                if (event.key.keysym.sym == SDLK_z)
                    return true;
            return false;
        }
    );

    strategy->setStrategy(     //Запуск через 10 секунд после завершения предыдущего
        [](float dt) {
            static float seconds = 0;
            seconds += dt;
            if (seconds > 10) {
                seconds = 0;
                return true;
            }
            return false;
        }
    );
    level_manager.addLevel<Level_1, Level_2>();         //Добавляем уровни
    level_manager.addLevel<Level_3>();                  //упс, забыли)

    level_manager.setLevelChangeStrategy(strategy);     //Добавляем нашу стратегию
    level_manager.setLoadingScreen(new MyScreen);       //Добавляем экран
}
void Game::preload() {
    level_manager.start(getRenderer());    //Не забываем вызвать start
}

void Game::render() {
    LevelManager::getInstance().render();  //Отрисовываем уровень
}

void Game::handleEvents() {

    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        level_manager.onEvents(event);     //Обрабатываем события
        //.....................
    }
}

void Game::update(float dt) {
    if (!level_manager.isRunning())        //Лучше всего проверку сделать тут
        quit();
    level_manager.update(dt);
}
```

8. Коллизии. Менеджер коллизий. Двойная диспетчеризация.

Самое время поговорить про обработку столкновений. Вероятнее всего в вашей игре будут перемещающиеся объекты. А что они обычно любят делать? Правильно, сталкиваться.

Так вот, нашу задачу можно разбить на 2 пункта:
1. Определить столкновение.
2. Обработать его (изменить каким-либо образом состояние объектов).

В классе `PhysicalObject` есть метод:
`virtual void checkCollide(PhysicalObject *other)` – вы должны определить его в своих физических классах.

Так вот, у нас есть наш объект, относительно которого и был вызван метод – его тип мы знаем, а второй объект был передан как указатель на `PhysicalObject` – его реальный тип мы не знаем. Как же его узнать, и правильно обработать столкновение?

Тут на помощь приходит класс `CollisionMap`.
Он позволит вам хранить обработчики столкновений и получать к ним доступ.
Основа его работы – механизм двойной диспетчеризации – это когда необходимо вызвать функцию, виртуальную по отношению к 2 объектам, а не одному.

Для начала вам необходимо написать функции-обработчики столкновений ваших игровых объектов – например так:

```
//Все классы должны быть наследниками PhysicalObject!

void tankTank(TankPhysics *tank_1, TankPhysics *tank_2) { /* .......... */ }

void tankBullet(TankPhysics *tank, BulletPhysics *bullet) { /* .......... */ }

void bulletBullet(BulletPhysics *bullet_1, BulletPhysics *bullet_2) { /* .......... */ }


void bulletTank(BulletPhysics *bullet, TankPhysics *tank) {
    tankBullet(tank, bullet);
}
```

Последняя функция просто меняет аргументы местами и вызывает уже определенный выше обработчик – незачем писать еще один обработчик для тех же самых типов объектов.

Теперь необходимо их добавить в менеджер:

```
//Добавляем ВСЕ обработчики, как показано в примере:
CollisionMap::getInstance().addCollisionHandler(tankBrick);
```

Далее, чтобы получить обработчик вызываем метод `lookup`:

```
void TankPhysics::checkCollide(PhysicalObject *other) {
    //параметры метода lookup - указатели на наши объекты:
    auto collide_handler = CollisionMap::getInstance().lookup(this, other);

    if (collide_handler)//обязательно сделайте проверку на случай, если функция не найдена
        collide_handler(this, other);  //вызвали обработчик
}
```

Теперь пара советов.
Совет 1. Как я говорил выше, для начала нам необходимо определить столкновение. Если не использовать никакие дополнительные средства, то придется проверять столкновения в самих обработчиках, которые вы добавляете в `CollisionMap`. Но, как вы можете догадаться, искать на каждой итерации обработчик в ассоциативном контейнере – **достаточно долго**.

Поэтому хорошо бы сделать следующее – каким-либо образом определить, что коллизия произошла вне обработчика, и только тогда искать и вызывать обработчик. В обработчике, соответственно, останутся только действия, которые необходимо проделать с УЖЕ столкнувшимися объектами. Один из способов – **Box2D**. В конце книги будет пример, но если кратко, то внутри метода `checkCollide` надо перебрать все произошедшие коллизии, и, если среди них есть пара наших тел, то искать и вызывать обработчик, т.е. как-то так:

```
void checkCollide(PhysicalObject *other) {
    for (b2ContactEdge* edge = /* ... */; edge; edge = edge->next) {
        if (edge->contact->IsTouching()) {   //пара столкнувшихся тел
           //Проверяем, что это были именно наши тела, т.е. this и other
           Если да, то:
           auto collide_handler = CollisionMap::getInstance().lookup(this, other);
           if (collide_handler) {
              collide_handler(this, other);
           }
    }
}
```

Совет 2. Можно еще ускорить процесс проверки – делать проверку коллизий только если **AABB** прямоугольники наших тел пересекаются (или какие-либо другие способы, которые позволят не проверять более детально точно не столкнувшиеся тела).

ПРИМЕРЫ.

1. Базовая функция для отрисовки.

Возможно, в процессе разработки игры выяснится, что отрисовка ваших текстур – одинаковая: просто задать размер, координаты, угол поворота. Тогда, почему бы не написать одну функцию, и вызывать ее из методов `draw` с подходящими параметрами?

```
void drawFrame(SDL_Renderer *renderer, double2 center, double angle,
double width, double height, const std::string &texture_id, int current_frame) {
    //Получить текстуру
    Texture texture = TextureManager::getInstance()->get(texture_id);
    int frame_width = texture.getWidth();
    int frame_height = texture.getHeight();

    //Считаем размеры в пикселах (зачем конвертация – написано выше)
    int on_screen_width = convert(width);
    int on_screen_height = convert(height);
    angle *= 180.0 / PI;     //Переводим из радиан в градусы

    //Координаты верхней левой точки квадрата будущего изображения - тоже конвертируем
    SDL_Point up_left_corner = convert({
        static_cast<int>(center.x - width / 2.0),
        static_cast<int>(center.y - height / 2.0)
    });

    //Берем конкретный кадр (обсуждалось ранее)
    SDL_Rect src_rect {
        current_frame * frame_width,
        0, frame_width, frame_height
    };

    SDL_Rect dest_rect {
        up_left_corner.x, up_left_corner.y,
        on_screen_width, on_screen_height
    };

    SDL_RenderCopyEx(
        renderer, texture,
        &src_rect, &dest_rect, angle,
        NULL, SDL_FLIP_NONE
    );
}
```

Напоминаем, в **SDL** – начало координат – верхний левый угол, ось OX вправо, OY вниз, а углы считаются по часовой стрелке начиная с оси OX.

2. Использование Box2D для определения коллизий.

Несколько несложных шагов позволят вам не определять коллизии самим.
Шаг 1. Создать переменную типа `b2World`:

```
b2Vec2 gravity{0.0, 0.0};
b2World collide_world(gravity);
```

Чтобы ее создать, нужно указать вектор гравитации. Т.к. у меня была top-view игра, я задал для гравитации нулевой вектор (т.е. без гравитации). Можно сделать эту переменную глобальной, для удобства.

Шаг 2. Добавить наши игровые объекты в “мир” Box2D. Советую добавить в ваши реализации класса `PhysicalObject` поле `b2Body *body` и в конструкторе проинициализировать его (подробное объяснение не приводится, т.к. можно прочитать в документации к **Box2D)**:

```
extern b2World collide_world;

class BasicPhysics : public PhysicalObject {
    b2Body *body;
    double length;
    double width;
public:
    BasicPhysics(const double2 &center_, double angle_,
    double length_, double width_, const b2BodyType &type_)
    : length(length_), width(width_) {
        b2PolygonShape shape;
        shape.SetAsBox(length / 2.0, width / 2.0);

        b2BodyDef bdef;
        bdef.position.Set(center_.x, center_.y);   //где будет наше тело
        bdef.angle = angle_;                       //и угол его поворота
        bdef.type = type_;                         //b2_staticBody или b2_dynamicBody
        bdef.bullet = false;

        body = collide_world.CreateBody(&bdef);
        body->CreateFixture(&shape, 1);
        body->SetUserData(this);   //чтобы мы потом могли идентифицировать наше тело
                                   //добавляем указатель this в качестве данных
    }

    //геттер, чтобы наследники могли получить доступ
    b2Body* getBody() {
        return body;
    }
};
```

Сначала мы создаем полигон – в данном случае задаем его просто прямоугольником. А далее создаем тело на его основе.

Не забудьте потом удалить тела из мира:

```
void destroyBody() { //если нужно удалить до вызова деструктора
    body->GetWorld()->DestroyBody(body);
    body = nullptr;  //чтобы не удалить в деструкторе еще раз
}

~BasicPhysics() {
    if (body)       //если еще не удалили
       destroyBody();
}
```

Шаг 3. Проверка столкновения:

```
void checkCollide(PhysicalObject *other) override {
    for (b2ContactEdge* edge = this->getBody()->GetContactList(); edge; edge = edge->next)
    {
        //просматриваем все коллизии
        if (edge->contact->IsTouching()) {
            b2Fixture* fixtureA = edge->contact->GetFixtureA();
            b2Fixture* fixtureB = edge->contact->GetFixtureB();
            b2Body* bodyA = fixtureA->GetBody();
            b2Body* bodyB = fixtureB->GetBody();
            //проверяем, что это наша пара:
            if (bodyA->GetUserData() == this && bodyB->GetUserData() == other ||
                bodyB->GetUserData() == this && bodyA->GetUserData() == other) {
                //находим обработчик
                auto collide_handler = CollisionMap::getInstance().lookup(this, other);
                if (collide_handler)
                    collide_handler(this, other);
            }
        }
    }
}
```

3. Музыкальный плейер.

Возможно, вы захотите, чтобы на протяжении всей игры на фоне играл саундтрек из одной или нескольких песен. Можете взять за основу следующий класс:

```
class MusicPlayer {
    std::vector<Mix_Music*> soundtrack;
    int cur_index = 0;
public:
    MusicPlayer();
    MusicPlayer(std::initializer_list<std::string> list);
    void addSoundtrack(const std::string &name);
    void playSoundtrack();
    ~MusicPlayer() {}
};
```

И реализация:

```
MusicPlayer::MusicPlayer() {
    Mix_PauseMusic();
    Mix_VolumeMusic(MIX_MAX_VOLUME / 2.0);
}

MusicPlayer::MusicPlayer(const std::initializer_list<std::string> &list) try
    : MusicPlayer() {
    auto it = list.begin();
    while (it != list.end()) {
        addSoundtrack(*it);
        ++it;
    }
}
catch(std::exception &e) {
    std::cerr << e.what() << std::endl;
}

void MusicPlayer::addSoundtrack(const std::string &name) {
    try {
        //Работает с Mix_Music, т.к. она для этого и предназначена
        Mix_Music *music = SoundManager<Mix_Music>::getInstance()->get(name);
        soundtrack.push_back(music);
    }
    catch (std::exception &e) {
        std::cerr << e.what() << std::endl;
    }
}

void MusicPlayer::playSoundtrack() {
    if (soundtrack.empty())
        return;
    if (!Mix_PlayingMusic()) {
        Mix_PlayMusic(soundtrack[cur_index], 1);
        cur_index = (cur_index + 1) % soundtrack.size();  //repeat playlist
    }
}
```

Можно добавить управление звуком, например переключатель громкости:

```
//diff может быть как положительным (увеличиваем), или отрицательным (уменьшаем)
void MusicPlayer::addVolume(int diff) {
    int new_volume = getVolume() + diff;
    if (new_volume < 0) {
        setVolume(0);
        std::cout << getVolume() << std::endl;
        return;
    }
    setVolume(new_volume);
    std::cout << getVolume() << std::endl;
}

int MusicPlayer::setVolume(int new_volume) const {
    return Mix_VolumeMusic(new_volume);
}

int MusicPlayer::getVolume() const {
    return Mix_VolumeMusic(-1);
}
```

Ну и управление **play/pause**:

```
bool MusicPlayer::isPaused() const {
    return Mix_PausedMusic();
}

void MusicPlayer::pause() {
    Mix_PauseMusic();
}

void MusicPlayer::resume() {
    Mix_ResumeMusic();
}
```

Так как канал для проигрывания музыки всего один, то можно сделать этот класс синглтоном, или просто создать в **main** или любом другом месте, откуда удобнее управлять:

```
int SDL_main(int argc, char ** argv) {
    {
        //.................
        MusicPlayer player = {"sound1", "sound2"};
        bool quit = false;
        while (/* ... */) {    //основной цикл
            player.playSoundtrack();
        }
    }
    finalize();
    return EXIT_SUCCESS;
}
```

4. Счетчик fps.

Возможно, вы в своей игре захотите ограничить fps значением, равным частоте обновления вашего монитора.
Воспользуетесь классом `FpsCounter`:

```
//В конструкторе укажите частоту кадров:
Game::Game() : fps_counter(60.0f) {}

//Поместите в самое начало метода render:
void Game::render() {
    if (!fps_counter.accept())
        return;
    //.................
}

//Так же можно получить текущее значение fps с помощью метода GetFps()
```